W4118: interrupt and system call

Junfeng Yang

Outline

- Motivation for protection
- Interrupt
- System call
Need for protection

- Kernel privileged, cannot trust user processes
  - User processes may be malicious or buggy

- Must protect
  - User processes from one another
  - Kernel from user processes
Hardware mechanisms for protection

- **Dual model of operation**
  - Privileged (+ non-privileged) operations in kernel mode
  - Non-privileged operations in user mode

- **Memory protection**
  - Segmentation and paging
    - E.g., kernel sets *page table* when creating process

- **Timer interrupt**
  - Kernel periodically gets back control
What operations are privileged?

- Read raw keyboard input
- Call `printf()`
- Call `write()`
- Write global descriptor table
- Divide by 0
- Set timer interrupt handler
- Set segment registers
- Load cr3
x86 protection modes

- Four modes (0-3), but often only 0 & 3 used
  - Kernel mode: 0
  - User mode: 3
  - “Ring 0”, “Ring 3”

- Segment has Descriptor Privilege Level (DPL)
  - DPL of kernel code and data segments: 0
  - DPL of user code and data segments: 3

- Current Privilege Level (CPL) = current code segment’s DPL
  - Can only access data segments when CPL <= DPL
Outline

- Motivation for protection
- Interrupt
- System call
OS: “event driven”

- **Events causing mode switches**
  - System calls: issued by user processes to request system services
  - Exceptions: illegal instructions (e.g., division by 0)
  - Interrupts: raised by devices to get OS attention

- **Often handled using same hardware mechanism**: interrupt
  - Also called trap
Interrupt view of CPU

while (fetch next instruction) {
    run instruction;
    if (there is an interrupt) {
        process interrupt
    }
}
x86 interrupt view

while (fetch next instruction) {
    run instruction;
    if (there is an interrupt) {
        switch to kernel stack if necessary
        save CPU context and error code if any
        find OS-provided interrupt handler
        jump to handler
        restore CPU context when handler returns
    }
}

- Q1: how does hardware find OS-provided interrupt handler?
- Q2: why switch stack?
- Q3: what CPU context to save and restore?
- Q4: what does handler do?
Q1: how to find interrupt handler?

- Hardware maps interrupt type to interrupt number

- OS sets up **Interrupt Descriptor Table (IDT)** at boot
  - Also called interrupt vector
  - IDT is in memory
  - Each entry is an interrupt handler
  - OS lets hardware know IDT base
  - Defines all kernel entry points

- Hardware finds handler using interrupt number as index into IDT
  - \( \text{handler} = \text{IDT}[\text{intr\_number}] \)
x86 interrupt hardware (legacy)

The diagram illustrates the x86 interrupt hardware with the following components:

- **PIC (Programmable Interrupt Controller)**: Sends interrupt requests (INTERRUPT) to the CPU.
- **CPU**: Receives interrupt requests and handles them.
- **IDT (Interrupt Descriptor Table)**: Stores interrupt handlers and their descriptors.
- **Mask points**: Points to specific interrupt handlers in the IDT.

IRPs (Interrupt Request Pins) are connected to the PIC, which in turn sends an interrupt request (INTR) to the CPU. The CPU then looks up the interrupt handler (intr #) in the IDT, which is indexed by the IDT descriptor table register (idtr) in the CPU.
x86 interrupt numbers

- Total 256 number [0, 255]
- Intel reserved first 32, OS can use 224
  - 0: divide by 0
  - 1: debug (for single stepping)
  - 2: non-maskable interrupt
  - 3: breakpoint
  - 14: page fault
  - 64: system call in xv6
  - xv6 traps.h
Interrupt gate descriptor
- Code segment selector and offset of handler
- Descriptor Privilege Level (DPL)
  - To invoke "int x" in software, must have CPL <= DPL
- Trap or exception flag. If exception, hardware clears the IF flag in EFLAGS to disable further maskable interrupts

lidt instruction loads CPU with IDT base

xv6
- Handler entry points: vector.S
- Interrupt gate format: SETGATE in mmu.h
- IDT initialization: tvinit() & lidt() in trap.c
Q2: why switch stack?

- **Cannot** trust stack of user process!

- x86 hardware switches stack when interrupt handling requires user-kernel mode switch
  - That is, when CPL <= DPL of handler’s code segment

- Where to find kernel stack?
  - task gate descriptor has SS and ESP for interrupt
  - ltr loads CPU with task gate descriptor

- xv6 uses current process’s kernel stack
  - switchuvm() in vm.c
Q3: what CPU context to save and restore?

- x86 saves SS, ESP, EFLAGS, CS, EIP, Err code
- Restored by `iret`
- OS can save more context

When switch stack:
- SS
- ESP
- EFLAGS
- CS
- EIP
- Err code

For some exceptions:
Q4: what does interrupt handler do?

- **Typical steps**
  - Assembly to save additional CPU context
  - Invoke C handler to process interrupt
    - E.g., communicate with I/O devices
  - Invoke kernel scheduler
  - Assembly to restore CPU context and return

- **xv6**
  - Interrupt handler entries: `vector.S`
  - Saves & restore additional CPU context: `trapasm.S`
  - C handler: `trap.c`, `struct trapframe` in `x86.h`
Interrupt v.s. Polling

- Instead for device to interrupt CPU, CPU can poll the status of device
  - Intr: “I want to see a movie.”
  - Poll: for(each week){“Do you want to see a movie?”}

- Good or bad?
  - For mostly-idle device?
  - For busy device?
Outline

- Motivation for protection
- Interrupt
- System call
System call

- User processes cannot perform privileged operations themselves

- Must request OS to do so on their behalf by issuing *system calls*

- OS must validate system call parameters
System call dispatch

1. Kernel assigns system call type a system call number
2. Kernel initializes system call table, mapping system call number to functions implementing the system call
   - Also called system call vector
3. User process sets up system call number and arguments
4. User process runs int X
5. Hardware switches to kernel mode and invokes kernel’s interrupt handler for X (interrupt dispatch)
6. Kernel looks up system call table using system call number
7. Kernel invokes the corresponding function
8. Kernel returns by running iret (interrupt return)
syscall() {
    fn = syscalls[%eax]
    // do real work
} // sysfile.c

movl $SYS_write, %eax
int 64
ret    // usys.S

write(fd, buf, sz)

sys_write(...) {
    // do real work
} // sysfile.c

xv6 system call dispatch

User mode

Kernel mode

User program

syscalls table

sys_write

IDT

syscall

64
System call parameter passing

- Typical methods
  - Pass via registers (e.g., Linux)
  - Pass via user-mode stack (e.g., xv6)
  - Pass via designated memory region

- xv6 system call parameter passing
  - Arguments pushed onto user stack based on gcc calling convention
  - Kernel function uses special routines to fetch these arguments
    - syscall.c
    - Why?
xv6 system call naming convention

- Usually a library function `foo()` will do some work and then call a system call `sys_foo()`
  - `sys_foo()` implemented in `sys*.c`
  - Often wrappers to `foo()` in kernel

- System call number for `foo()` is `SYS_foo`
  - `syscalls.h`

- All system calls begin with `sys_`
Tracing system calls

- Use the “strace” command (man strace for info)

- Linux has a powerful mechanism for tracing system call execution for a compiled application

- Output is printed for each system call as it is executed, including parameters and return codes

- ptrace() system call is used to implement strace
  - Also used by debuggers (breakpoint, singlestep, etc)

- Use the “ltrace” command to trace dynamically loaded library calls